Homework5

Charles Zhang

```
#The following line sources functions from the class file `365Functions.r`.
source("https://drive.google.com/uc?export=download&id=10dNH3VbvxS8Z3OHjP4i9gRbtsf91VVBb")
require(Matrix)
require(ggplot2)
```

Problem 1

Note: The first three parts of this question are from the interpolating polynomials activity

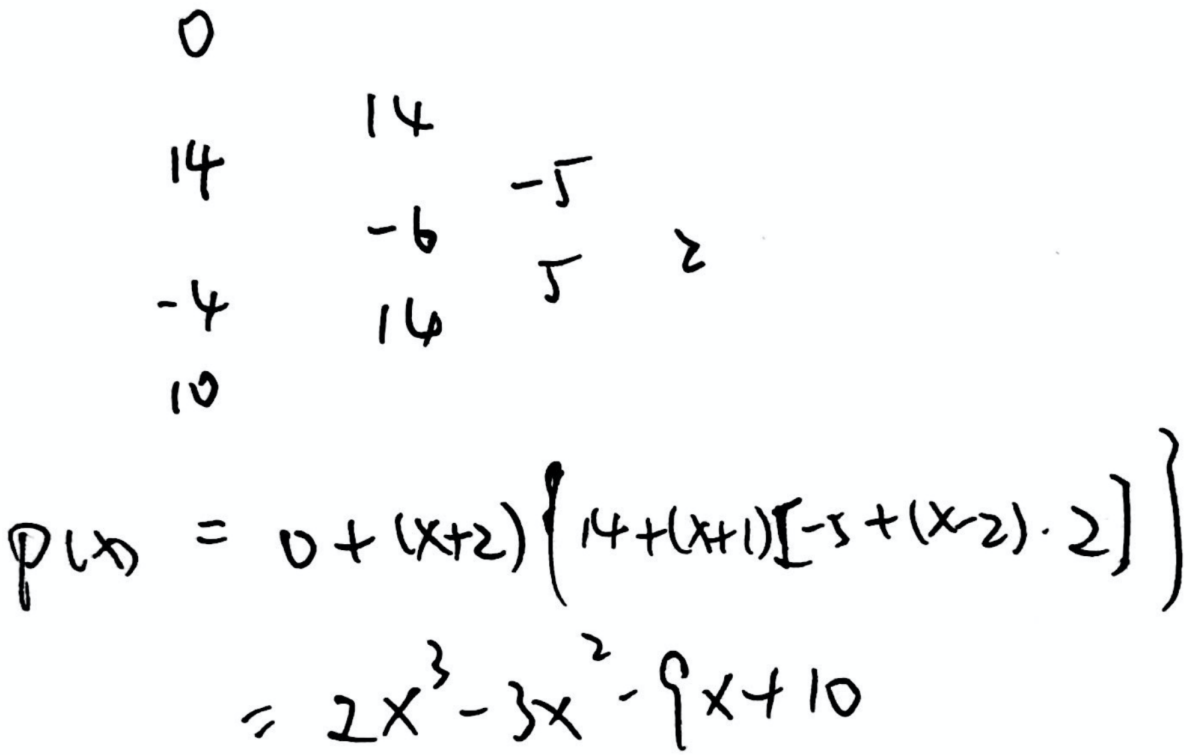
Consider the points (-2,0), (-1,14), (2,-4), (3,10). Find the degree 3 (i.e., the highest order term is cx^3) interpolating polynomial p(x) for these points in three different ways.

a) Calculate the Vandermonde matrix and solve the equation for the coefficients using `R`'s built-in `solve` function.

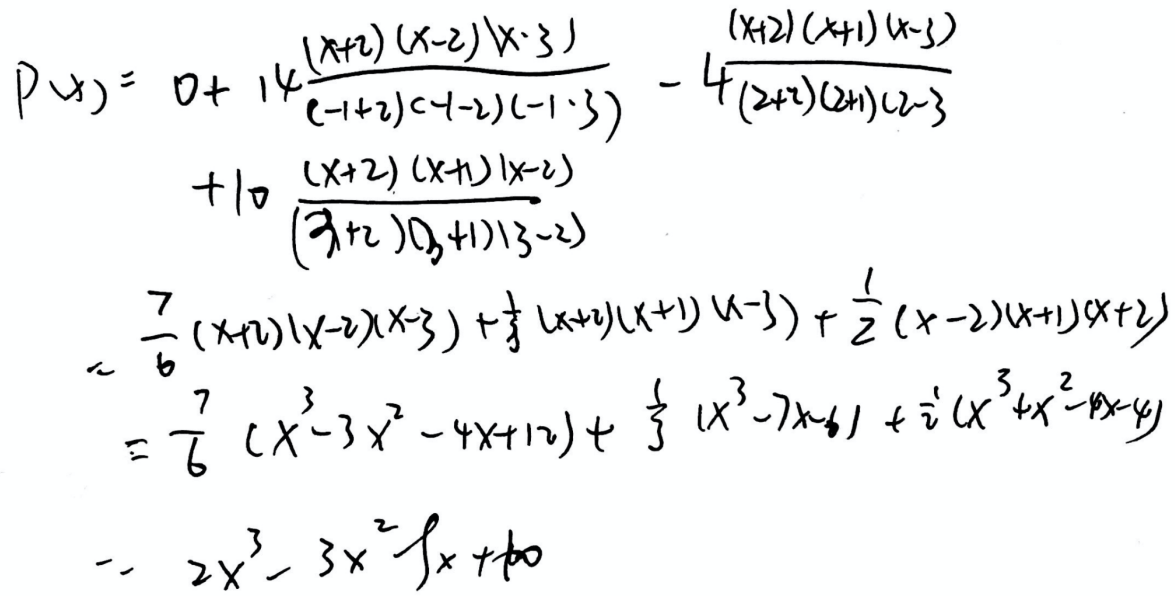
```
x=c(-2,-1,2,3)
y=c(0,14,-4,10)
V=Vandermonde(x)
(c=solve(V,y))
```

```
## [1] 10 -9 -3  2
```

b) By hand, find the interpolating polynomial using Newton's divided differences. You need to multiply it out to see that you are getting the same answer as above.



c) By hand, find the interpolating polynomial using Lagrange interpolation. Yes this is a pain. Yes it is worth doing this one time in your life to really feel in your gut that it works.



d) In `365Functions.r`, I have included a function `Interpolator` that does polynomial interpolation using either the Vandermonde matrix or Newton's divided difference method. In theory, these two methods should yield the same interpolating polynomial (which is unique as long as the interpolation points are at different x values). Have a quick look at my code to understand what it is doing. Then find an example where the two methods give two different answers or one method finds an answer and the other does not. Explain the discrepancy.

```
x = seq(1, 15)
y = runif(15)
xx = seq(1, 15)
Interpolator(x, y, xx)
```

```
##  [1] 0.56991196 0.95120984 0.21909142 0.54552544 0.73674112 0.74159800
##  [7] 0.10750962 0.44739294 0.80499757 0.68674285 0.82141297 0.71262793
## [13] 0.09660268 0.36890194 0.95416167
```

```
# Interpolator(x, y, xx, Itype = "Vandermonde")
```

The result can not get since the condition number = 4.26216e-32.

```
v = Vandermonde(x)
Cond(v)
```

```
## [1] 9.983507e+19
```

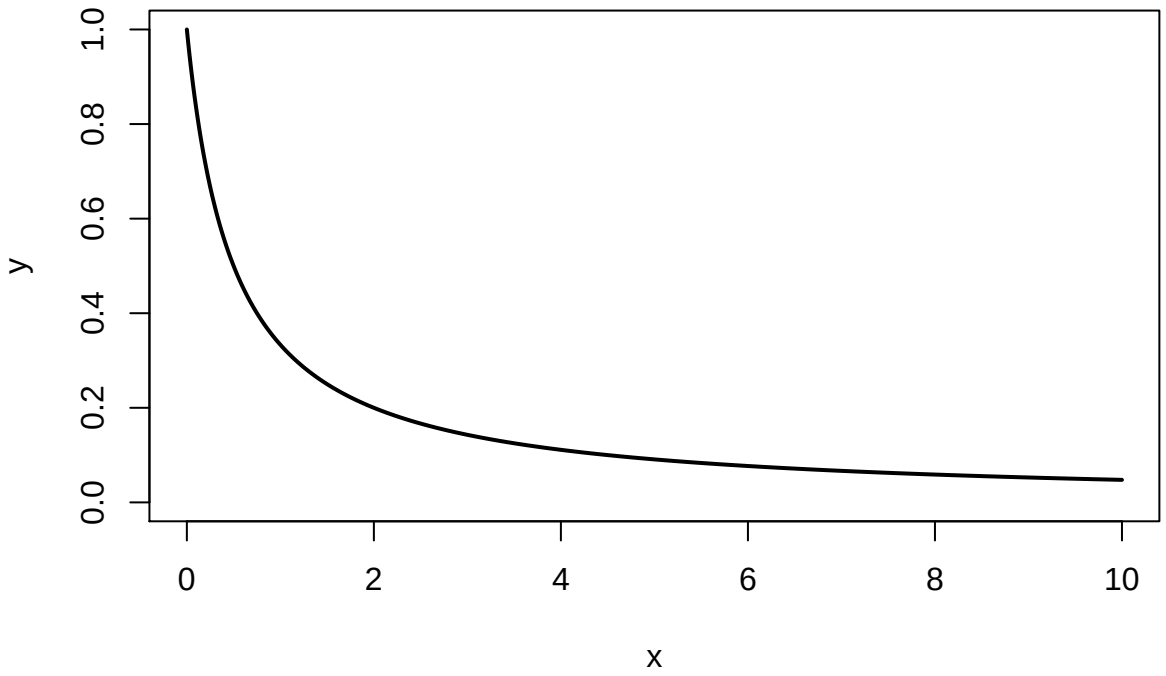
Newton's divided difference can solve the polynomial, but Vandermonde cannot, since the Vandermonde matrix is ill-conditioned.

Problem 2

***Note: This was the activity on Chebyshev polynomials**

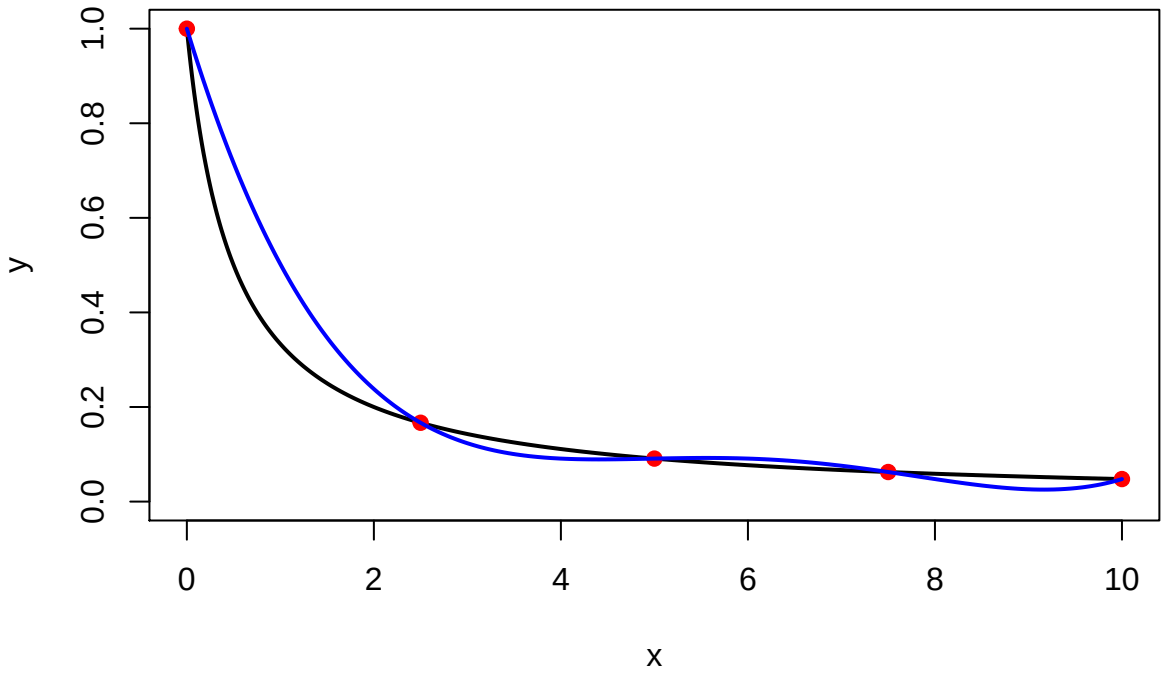
Here is the function $f(x) = \frac{1}{1+2x}$ on the interval $[0, 10]$:

```
f=function(x){1/(1+2*x)}
a = 0
b = 10
xx = seq(a,b,length=1000)
yf=f(xx)
plot(xx,yf,type='l',xlab="x",ylab="y",xlim=c(a,b),ylim=c(0,1),col='black',lwd=2)
```



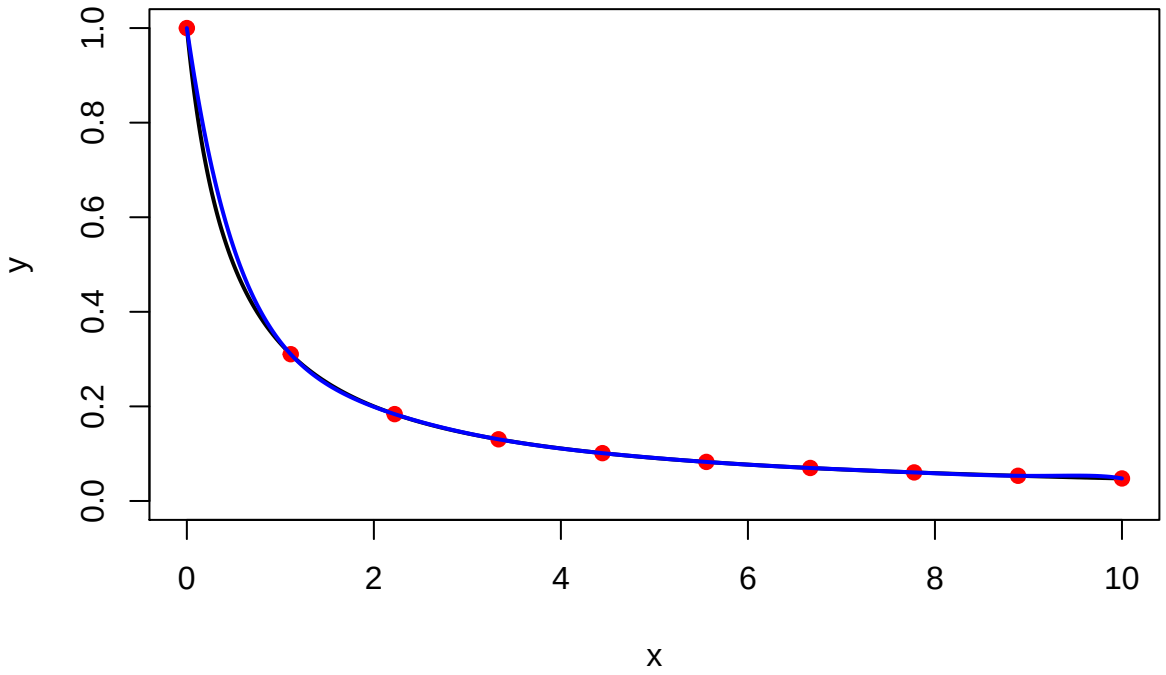
a) Use 5 evenly spaced nodes on the interval [0,10] to generate a polynomial interpolant $p_{e5}(x)$ of $f(x)$ on this interval (you can again use my `Interpolator` function). Make a plot with the graph of the function $f(x)$ in black, the graph of the interpolating polynomial in blue, and the 5 interpolating points in red.

```
xi= seq(a,b,length=5)
yi = f(xi)
y5 = Interpolator(xi,yi,xx)
plot(xx,yf,type='l',xlab="x",ylab="y",xlim=c(a,b),ylim=c(0,1),col='black',lwd=2)
points(xi, yi, pch=19, col='red')
lines(xx,y5,col='blue',lwd=2)
```



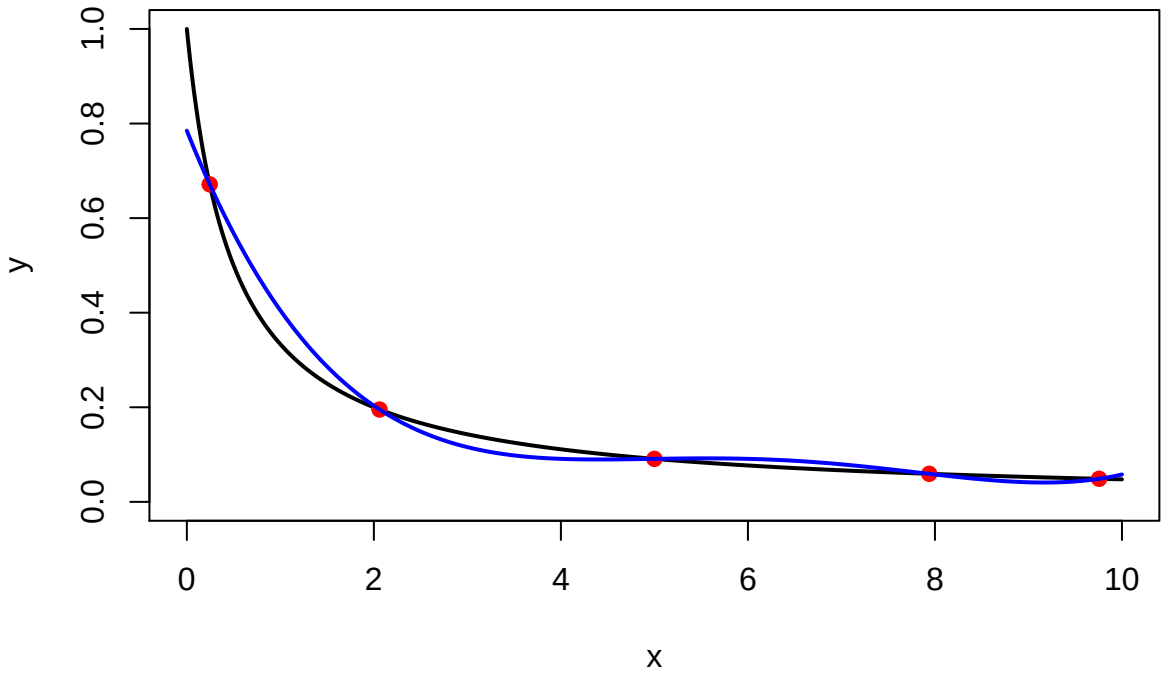
b) Repeat part a) with 10 evenly spaced nodes on the interval [0,10].

```
xi= seq(a,b,length=10)
yi = f(xi)
y10 = Interpolator(xi,yi,xx)
plot(xx,yf,type='l',xlab="x",ylab="y",xlim=c(a,b),ylim=c(0,1),col='black',lwd=2)
points(xi, yi, pch=19, col='red')
lines(xx,y10,col='blue',lwd=2)
```



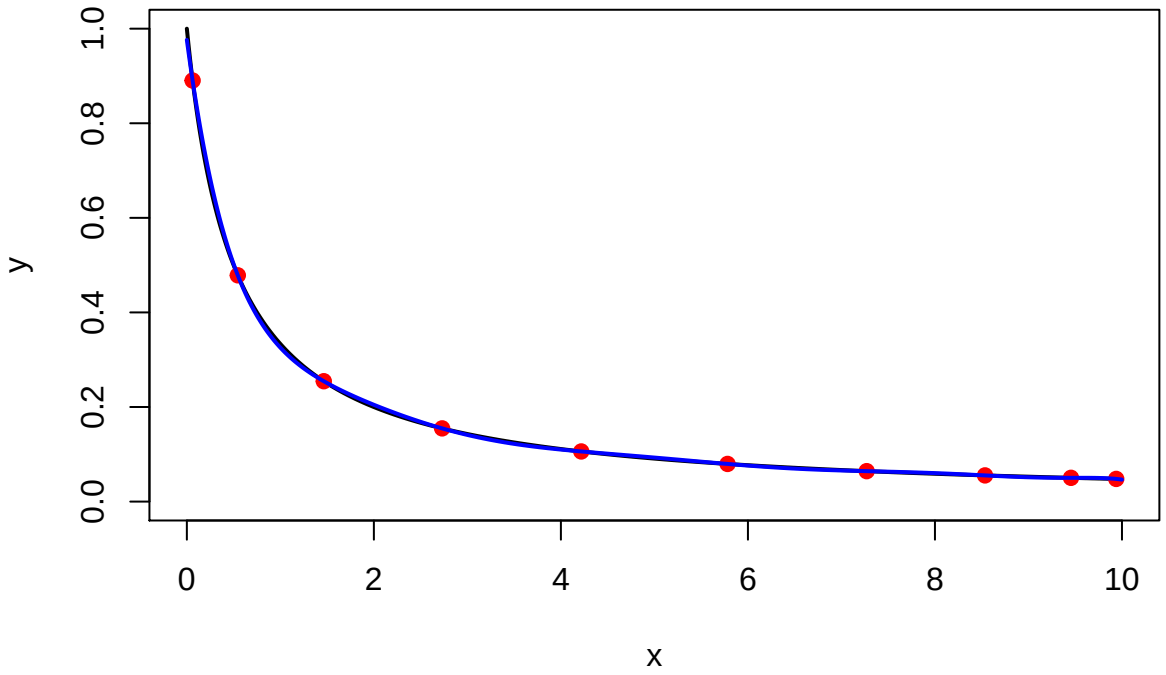
c) Find the zeros of a degree 5 Chebyshev polynomial (shifted and stretched to cover the interval [0,10]). Repeat part a) using these 5 nodes to generate a polynomial interpolant $p_{c5}(x)$ of $f(x)$ on this interval (you can again use my `Interpolator` function).

```
n = 5
Cxi = (b-a)/2*cos((2*(1:n)-1)*pi/(2*n)) + (b+a)/2
Cyi = f(Cxi)
yC5 = Interpolator(Cxi,Cyi,xx)
plot(xx,yf,type='l',xlab="x",ylab="y",xlim=c(a,b),ylim=c(0,1),col='black',lwd=2)
points(Cxi,Cyi,pch=19,col='red')
lines(xx,yC5,col='blue',lwd=2)
```



d) Repeat part c) with the zeros of a degree 10 Chebyshev polynomial.

```
n = 10
Cxi10 = (b-a)/2*cos((2*(1:n)-1)*pi/(2*n)) + (b+a)/2
Cyi10 = f(Cxi10)
yC10 = Interpolator(Cxi10,Cyi10,xx)
plot(xx,yf,type='l',xlab="x",ylab="y",xlim=c(a,b),ylim=c(0,1),col='black',lwd=2)
points(Cxi10,Cyi10,pch=19,col='red')
lines(xx,yC10,col='blue',lwd=2)
```



e) Compare the errors $||f - p||_\infty$ for each of these four approximating polynomials on the interval $[0, 10]$. You can approximate these errors by finding the maximum absolute value of the differences between the function and the approximating polynomial on the 1000 evenly spaced points you used to plot the function. Here is an example:

```
err5 = yf - y5
print("5 Evenly Distributed Nodes Error:")
```

```
## [1] "5 Evenly Distributed Nodes Error:"
```

```
print(max(abs(err5)))
```

```
## [1] 0.2159372
```

```
err10 = yf - y10
print("10 Evenly Distributed Nodes Error:")
```

```
## [1] "10 Evenly Distributed Nodes Error:"
```

```
print(max(abs(err10)))
```

```
## [1] 0.05378002
```

```
errC5 = yf - yC5
print("Chebyshev Degree 5 Error:")
```

```
## [1] "Chebyshev Degree 5 Error:"
```

```
print(max(abs(errC5)))
```

```
## [1] 0.2151389
```

```
errC10 = yf - yC10
print("Chebyshev Degree 10 Error:")
```

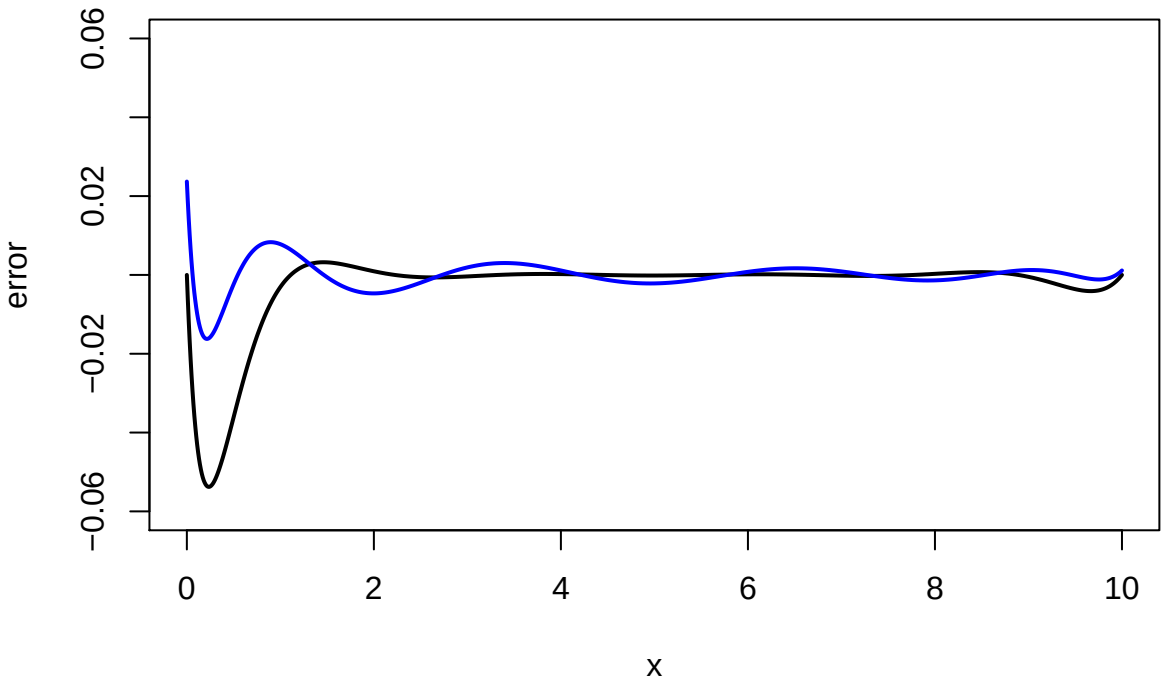
```
## [1] "Chebyshev Degree 10 Error:"
```

```
print(max(abs(errC10)))
```

```
## [1] 0.02369063
```

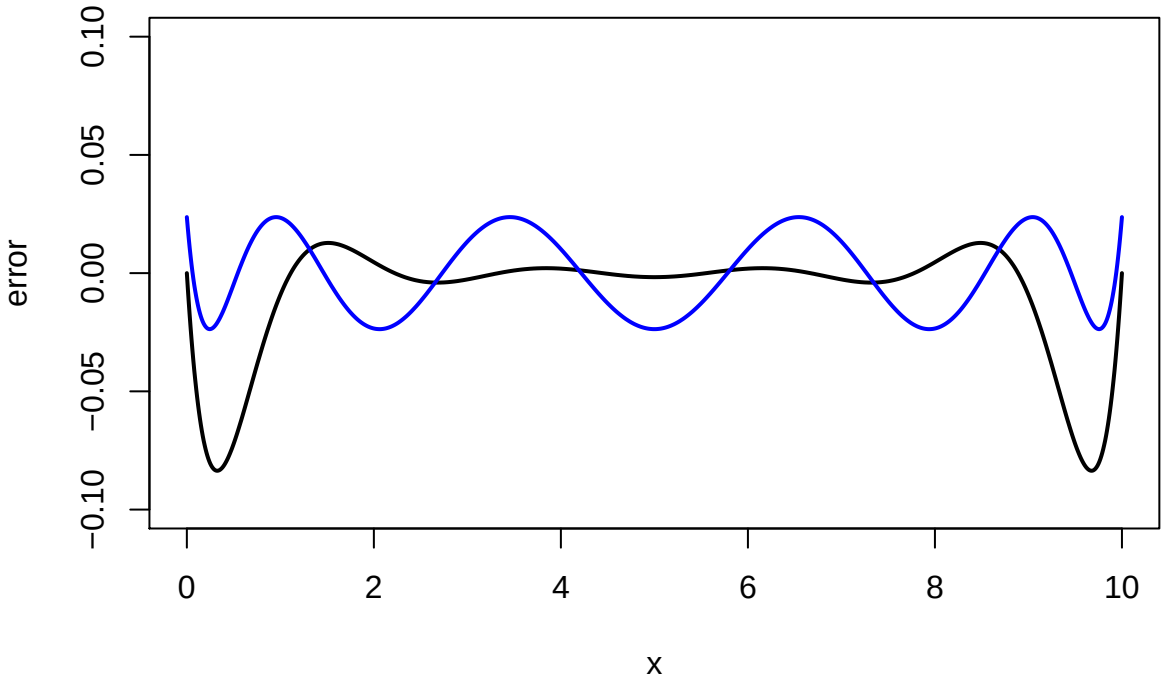
f) Show the errors from parts b) and d) on a single plot. Comment on the plot.

```
plot(xx,err10, type='l', xlab="x",ylab="error",xlim=c(a,b),ylim=c(-.06,.06),col='black',lwd=2)
lines(xx, errC10, type='l', col='blue',lwd=2)
```



g) On a single plot, show the errors from parts b) and d) divided by the function values `yf`. Comment on the plot.

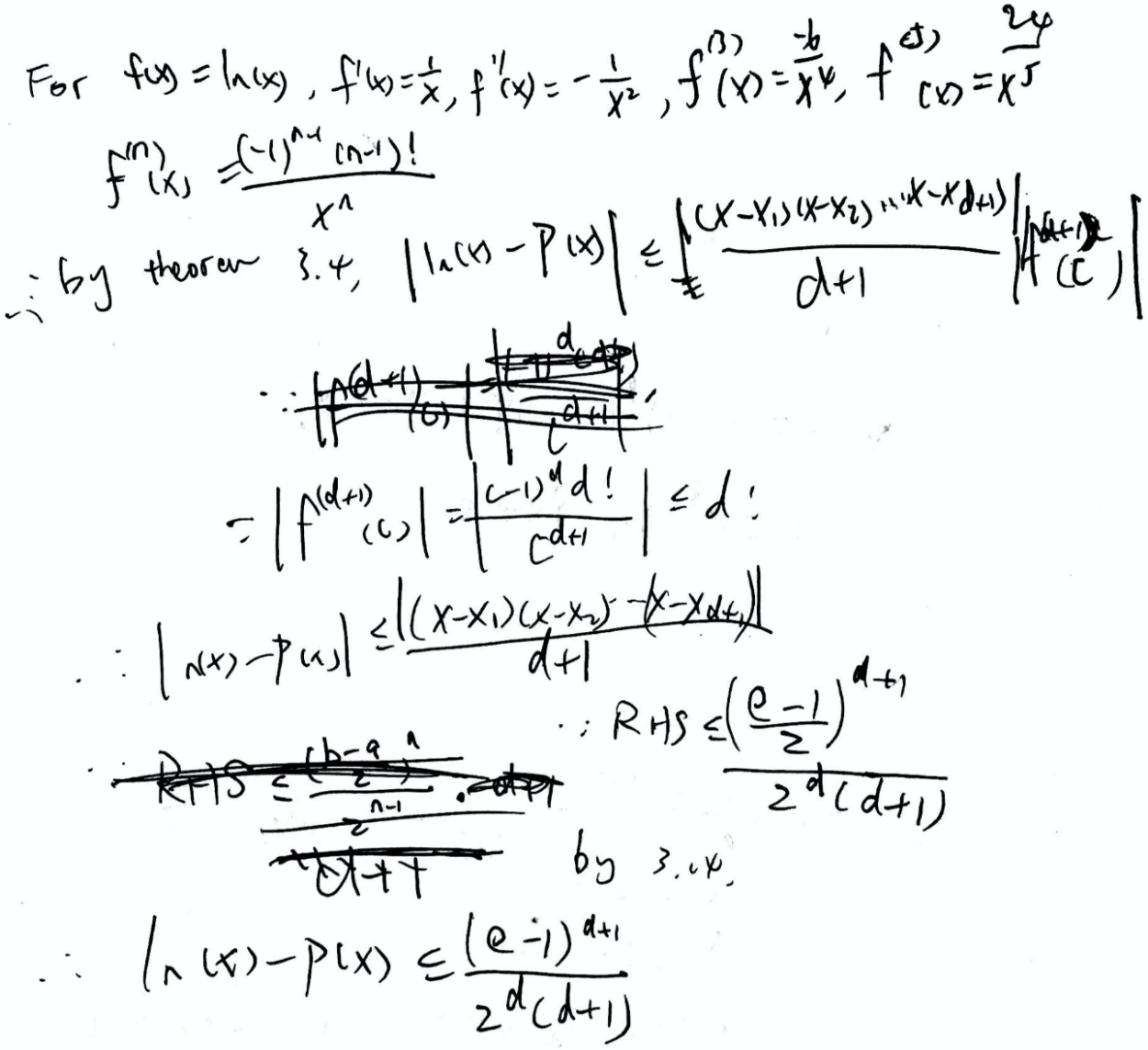
```
plot(xx,err10/yf, type='l', xlab="x",ylab="error",xlim=c(a,b),ylim=c(-.1,.1),col='black',lwd=2)
lines(xx, errC10/yf, type='l', col='blue',lwd=2)
```



Problem 3

Suppose you are designing a natural log (`ln`) key for a calculator that displays 5 digits to the right of the decimal point. Find the least degree d for which Chebyshev interpolation on the interval $[1, e]$ will always approximate the natural log function to 5 digits of accuracy (i.e., the interpolation error will always be less than $0.5 * 10^{-5}$)?

Hints: You will need to compute the derivatives of the natural log function, and use the error estimates in both equation (3.14) and Theorem 3.4 of the book.



```
for (d in 1:50) {
  err = (exp(1)-1)^(d+1)/(2^(2*d+1))/factorial(d+1)*factorial(d)
  if (err <= 0.5 * 10 ^-5) {
  cat("Degree:", d, "\n")
  cat("Error:", err)
  break
  }
}
```

```
## Degree: 12
## Error: 2.60936e-06
```

Take d = 12 to get the right hand side less than 0.5 * 10 ^-5

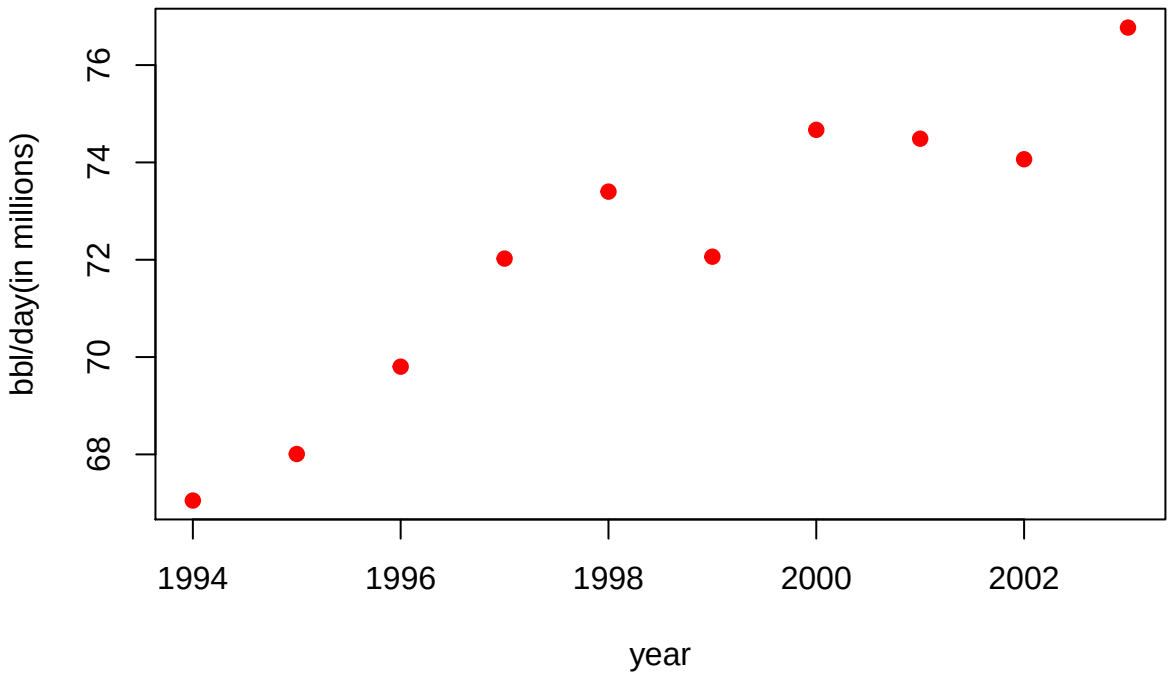
Problem 4

Here is the data from Computer Problem 3.2.3 in the book on total world oil production in millions of barrels per day between 1994 and 2003.

```
year = seq(1994,2003)
bbld = c(67.052,68.008,69.803,72.024,73.400,72.063,74.669,74.487,74.065,76.77)
print(cbind(year,bbld))
```

```
##       year   bbld
##  [1,] 1994 67.052
##  [2,] 1995 68.008
##  [3,] 1996 69.803
##  [4,] 1997 72.024
##  [5,] 1998 73.400
##  [6,] 1999 72.063
##  [7,] 2000 74.669
##  [8,] 2001 74.487
##  [9,] 2002 74.065
## [10,] 2003 76.770
```

```
plot(year,bbld,col='red',pch=20,cex=1.5,ylab="bbl/day(in millions)")
```



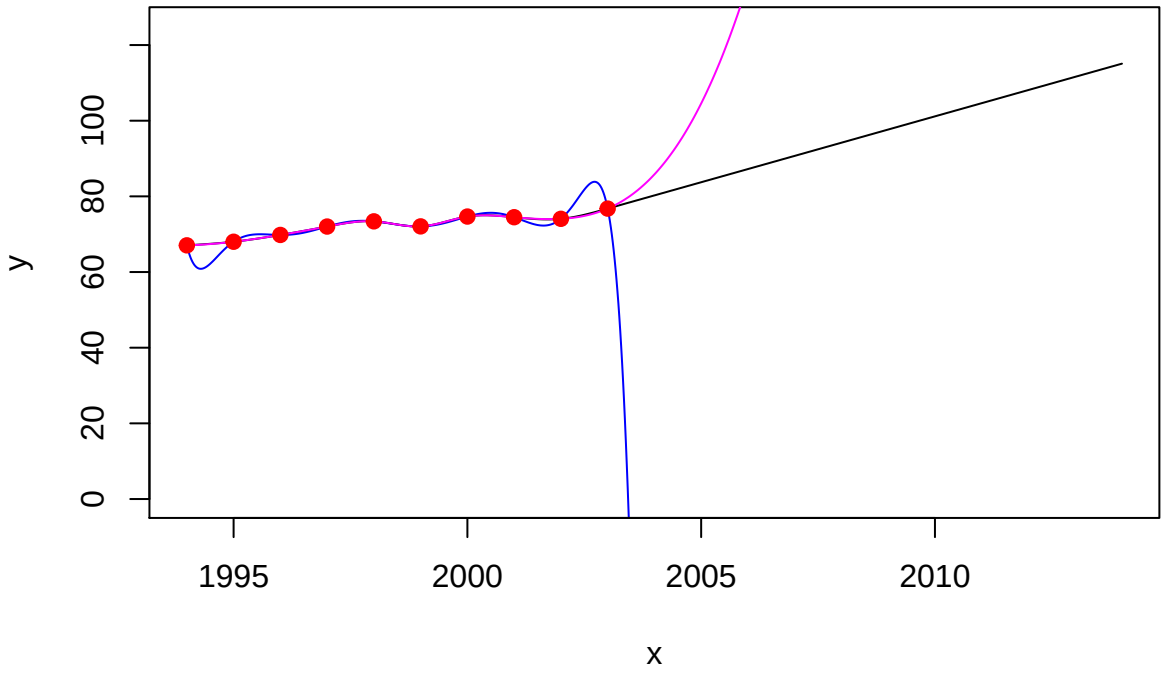
Our objective is to interpolate a function to fit this data and then extrapolate that function to predict the total world oil production in 2010.

a) Generate three different interpolating functions and plot all three functions on the same graph, with a range of 1994 to 2014.:

- the exact polynomial interpolant (you can use my `Interpolator` function)
- a natural cubic spline (you can use R's built-in `splinefun` function with `method="natural"`)
- a not-a-knot cubic spline (you can use R's built-in `splinefun` function with `method="fmm"`)

```
x = seq(1994,2014,len=1000)
c=NewtonDD(year,bbld)
```

```
py = Horner(c,x,year)
ns = splinefun(year,bbld,method='natural')
nsy=ns(x)
fs = splinefun(year,bbld,method='fmm')
fsy=fs(x)

plot(x,py,type='l',col='blue',ylim=c(0,125),xlab='x',ylab='y')
lines(x,nsy,type="l",col="black")
lines(x,fsy,type="l",col="magenta")
points(year,bbld,col='red',pch=20,cex=1.5)
```



b) Compute the three different predicted values for total world oil production in millions of barrels per day in 2010.

```
Horner(c,2010,year)
```

```
## [1] -1951726
```

```
ns(2010)
```

```
## [1] 101.1491
```

```
fs(2010)
```

```
## [1] 461.6587
```

c) *Briefly* state which method you think is the best of the three, and why.

The natural cubic spline is the best for prediction since it is the linear extrapolation based on the final two points

d) What was the actual total world oil production in 2010 (use google)?

87.16 million barrels per day.

Problem 5

This was a previous take-home exam question.

In Technical Report 1, we explored numerical differentiation via finite difference approximations. In this question, we are going to explore numerical integration (also called *quadrature*) using ideas we've learned about function approximation and polynomial interpolation. We want to approximate the integral

$$\int_a^b f(x)dx.$$

We'll consider three different methods. The main idea behind the first two methods is that we are going to approximate the function $f(\cdot)$ by a polynomial of degree $n-1$ passing through n sampled points of the function. We'll compare two different choices for the x-values of the points: (i) the n roots of the Chebyshev polynomial $T_n(x)$, stretched and shifted to cover the interval $[a,b]$; and (ii) the x-values of the n extrema (minima and maxima) of the Chebyshev polynomial $T_{n-1}(x)$, stretched and shifted to cover the interval $[a,b]$. Note that the Chebyshev polynomial $T_n(x)$ has $n+1$ extrema, so that is why we are using $T_{n-1}(x)$ to find the locations of n extrema.

a) Write two functions `cheb.zeros(a,b,n)` and `cheb.extrema(a,b,n)` that return the desired x-values of n interpolation points for each of these methods.

Run these commands to test your functions by generating 5 points on the interval $[2,6]$:

```
cheb.zeros=function(a,b,n){
 r=sort(cos((2*(1:n)-1)*pi/(2*n)))
 s=r*(b-a)/2
return(s+(a+b)/2)
}

cheb.extrema=function(a,b,n){
extrema=sort(cos(pi*(0:(n-1)/(n-1))))
s=extrema*(b-a)/2
return(s+(a+b)/2)}
```

```
cheb.zeros(2,6,5)
```

```
## [1] 2.097887 2.824429 4.000000 5.175571 5.902113
```

```
cheb.extrema(2,6,5)
```

```
## [1] 2.000000 2.585786 4.000000 5.414214 6.000000
```

b) I've started a function `cheb.integrate(f,a,b,n,method)` for you that should take a function, an interval, a number of interpolation points, and the method (zeros or extrema), and returns an approximation of the definite integral $\int_a^b f(x)dx$. After calling your functions from part (a) to get the interpolation points, you need to (i) compute the coefficients of the interpolating polynomial, and (ii) use the coefficients to compute the integral of the interpolating polynomial over the interval $[a,b]$. For the first part of the task, you are allowed to use the Vandermonde matrix. Even though it is not particularly scalable and it would be better to use Newton's divided differences, it is more straightforward to code the interpolation with the Vandermonde matrix.

As an aside, in practice, neither the Vandermonde or Newton's divided difference methods are used. Rather, the integral is approximated by the sum $\sum_{i=0}^{n-1} w_i f(x_i)$, where the w_i's are some weights that can be calculated ahead of time in a fast manner.

For the second part of the task, check your work with your calculus knowledge about the closed form of an integral of a polynomial over an interval.

```
cheb.integrate=function(f,a,b,n,method='extrema'){
  if(method=='extrema'){
    x=cheb.extrema(a,b,n)
  }
  else if(method=='zeros'){
    x=cheb.zeros(a,b,n)
  }
  else{stop('Uknown method')}

  # Fill in your code here to (i) find the coefficients of interpolating polynomial and (ii) use them t
  # Hint: my code here is 4 lines

  v = Vandermonde(x)
  coeffs = solve(v, f(x))
  int.coeffs=c(0, coeffs/(1:n))
  return(Horner(int.coeffs,b) - Horner(int.coeffs,a))
}
```

Test your code on $\int_{-1}^{2} x^2 \, dx$ with 5 points:

```
f=function(x){x^2}
cheb.integrate(f,-1,2,5,method='extrema')
```

```
## [1] 3
```

```
cheb.integrate(f,-1,2,5,method='zeros')
```

```
## [1] 3
```

c) In the third method, called Simpson's Rule, we take n evenly spaced points (n should be an odd integer), with the first equal to a and the last equal to b. Then we create a quadratic spline, with one quadratic function fit through the points x_0, x_1, and x_2, the next fit through the points x_2, x_3, and x_4, and so forth. The figure below, which is from http://tutorial.math.lamar.edu/Classes/CalcII/ApproximatingDefIntegrals.aspx, shows a picture demonstrating this process with $n = 7$.

Note that the black lines on top of the shaded areas are the quadratic functions passing through three points of the red function, which is the function we are approximating. The first quadratic goes through the points $(x_0, f(x_0))$, $(x_1, f(x_1))$, and $(x_2, f(x_2))$, and the area under it is shaded in green. The second quadratic goes through the points $(x_2, f(x_2))$, $(x_3, f(x_3))$, and $(x_4, f(x_4))$, and the area under it is shaded in yellow. The third quadratic (the easiest to recognize as a quadratic) goes through the points $(x_4, f(x_4))$, $(x_5, f(x_5))$, and $(x_6, f(x_6))$, and the area under it is shaded in blue. The integral $\int_{x_0}^{x_6} f(x)dx$ is approximated by the sum of the green, yellow, and blue areas.

Using the Lagrange form of the interpolating quadratic between the first three points, we have

$$p_1(x) = f(x_0)\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + f(x_1)\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + f(x_2)\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}.$$

If we let $h = x_{i+1} - x_i$ (the distance between any two of the evenly spaced points), then this becomes

$$p_1(x) = f(x_0)\frac{(x-x_1)(x-x_2)}{2h^2} - f(x_1)\frac{(x-x_0)(x-x_2)}{h^2} + f(x_2)\frac{(x-x_0)(x-x_1)}{2h^2},$$

and

$$\begin{aligned}\int_{x_0}^{x_2} p_1(x)dx &= \frac{f(x_0)}{2h^2}\int_{x_0}^{x_2}(x-x_1)(x-x_2)dx - \frac{f(x_1)}{h^2}\int_{x_0}^{x_2}(x-x_0)(x-x_2)dx + \frac{f(x_2)}{2h^2}\int_{x_0}^{x_2}(x-x_0)(x-x_1)dx \\ &= \frac{h}{3}f(x_0) + \frac{4h}{3}f(x_1) + \frac{h}{3}f(x_2) \\ &= \frac{h}{3}\left[f(x_0) + 4f(x_1) + f(x_2)\right].\end{aligned}$$

So to approximate the integral, we can sum over each of the $\frac{n-1}{2}$ quadratic segments:

$$\int_a^b f(x)dx \approx \sum_{k=1}^{\frac{n-1}{2}} \int_{x_{2(k-1)}}^{x_{2k}} p_k(x)dx = \sum_{k=1}^{\frac{n-1}{2}} \frac{h}{3}\left[f(x_{2(k-1)}) + 4f(x_{2k-1}) + f(x_{2k})\right],$$

where k indexes the quadratic segments. Note that the right-hand side is just a linear combination of the function values at the n evenly spaced points!

Implement a function `simpson.integrate` that takes a function f, an interval $[a, b]$, and an odd number of points n, and approximates $\int_a^b f(x)dx$ by the sum on the right-hand side of the most recent equation above.

```
simpson.integrate=function(f,a,b,n){
if(n%%2==0)
  stop('n must be odd')
# fill in the rest of the function here
  sum = 0
  points = seq(a,b,length=n)
  vals = f(points)
  h = points[2] - points[1]
  for (i in 1:((n-1)/2)) {
    sum = sum + h/3*(vals[1+2*(i-1)] + 4*vals[2+2*(i-1)] + vals[3+2*(i-1)])
  }
  return(sum)
}
```

Test your `simpson.integrate` code on the integral $\int_0^1 x^{1.5}dx$, with $n = 7$ points (i.e., an approximation comprised of three quadratic functions concatenated):

```
g=function(x){x^1.5}
simpson.integrate(g,0,1,7)
```

```
## [1] 0.400158
```

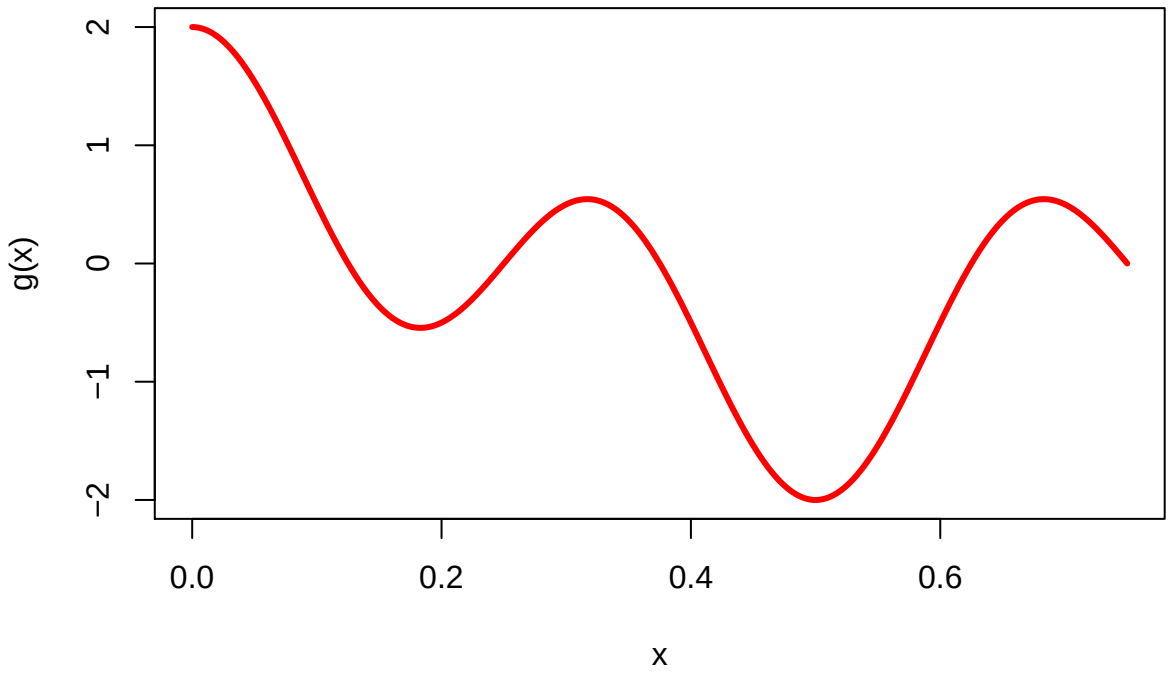
d) For any good numerical integration method, the magnitude of the error between the actual definite integral value and the approximation should decrease as the number of integration points/sub-intervals you use for the approximation increases. Here is a function to plot the magnitudes of the errors for different values of n and for each of the three methods you implemented above. The parameter `correct` is the actual value of the integral of f over the interval $[a, b]$. You can compute this analytically by hand (woohoo!) or using Wolfram Alpha.

```
integration.test=function(f,a,b,max.n,correct){
  nn.odd=seq(3,max.n,by=2)
  nn=2:max.n

  # Compute the errors
  # Note: I added machine epsilon to all errors so that the plots will still work if the actual error i
  errors.zeros=rep(NA,length(nn))
```

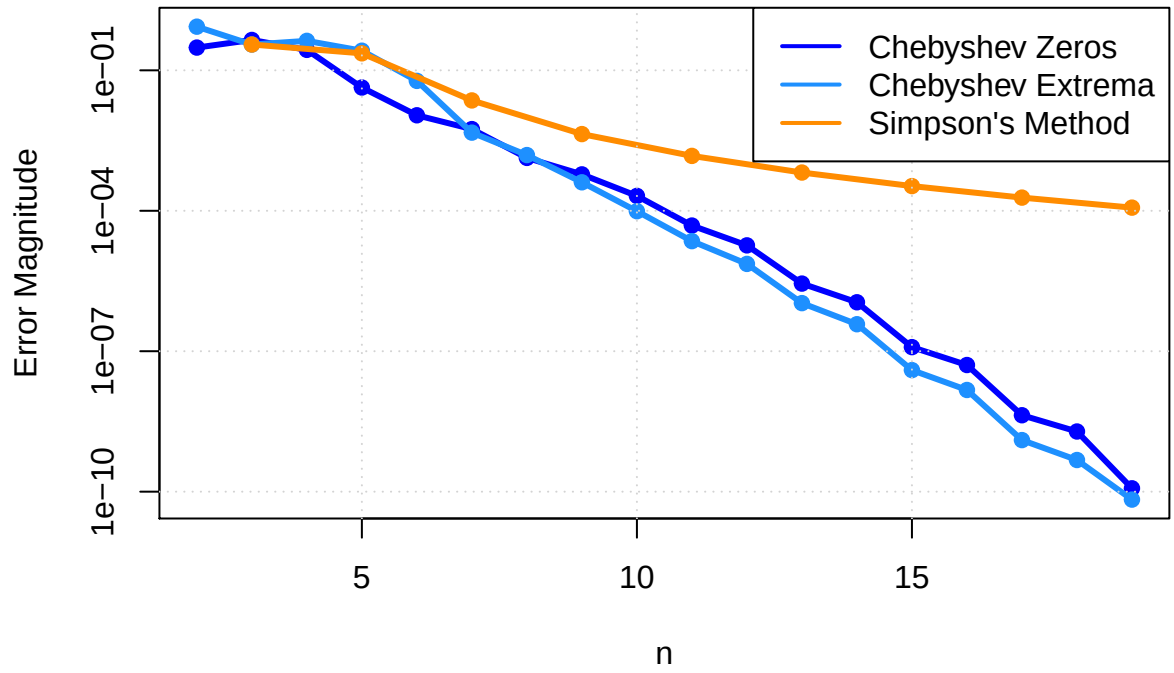
```
  errors.extrema=rep(NA,length(nn))
  errors.simpson=rep(NA,length(nn.odd))
  for (i in 1:length(nn)){
    errors.zeros[i]=abs(cheb.integrate(f,a,b,nn[i],method='zeros')-correct)+.Machine$double.eps
    errors.extrema[i]=abs(cheb.integrate(f,a,b,nn[i],method='extrema')-correct)+.Machine$double.eps
  }
  for (i in 1:length(nn.odd)){
    errors.simpson[i]=abs(simpson.integrate(f,a,b,nn.odd[i])-correct)+.Machine$double.eps
  }

  # Plot the error magnitudes
 plot(nn,errors.zeros,log="y",type="l",lwd=3,col="blue",xlab="n",ylab="Error Magnitude",ylim=range(erro
  points(nn,errors.zeros,log="y",pch=19,col="blue",)
  lines(nn,errors.extrema,log="y",type="l",lwd=3,col="DodgerBlue")
  points(nn,errors.extrema,log="y", pch=19,col="DodgerBlue",)
  lines(nn.odd,errors.simpson,log="y",type="l",lwd=3,col="DarkOrange")
  points(nn.odd,errors.simpson,log="y",pch=19,col="DarkOrange")
  grid()
  legend("topright",legend=c("Chebyshev Zeros", "Chebyshev Extrema", "Simpson's Method"),col=c("blue","
}
```

Here is an example:

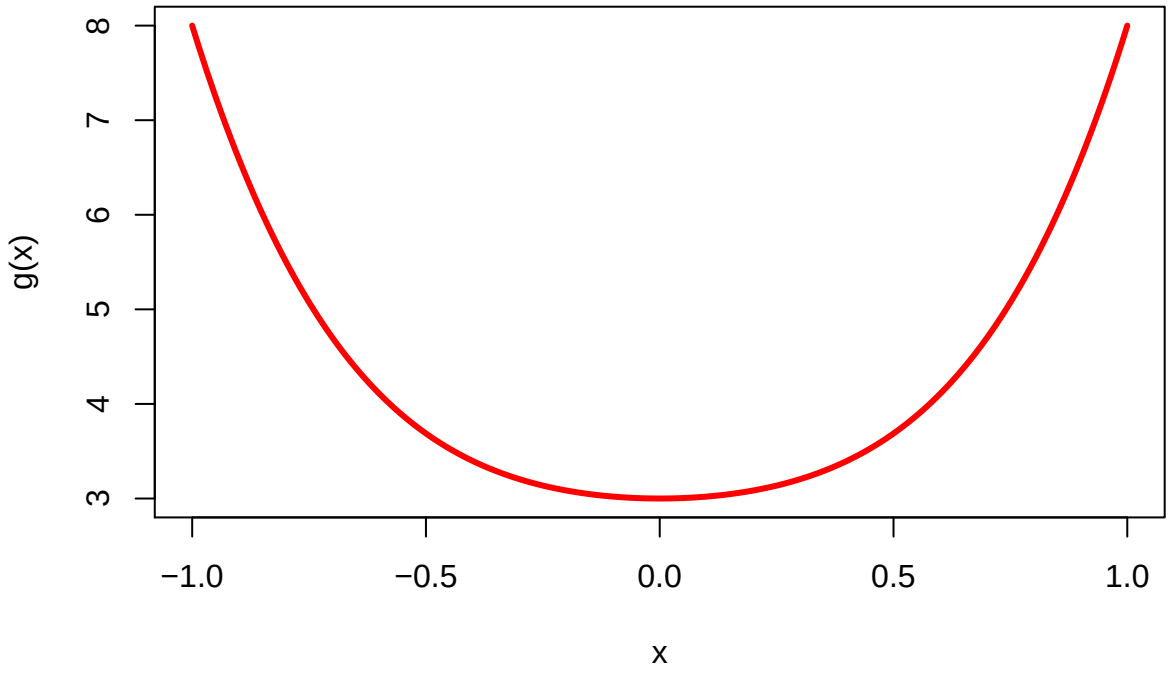
```
g=function(x){cos(2*pi*x)+cos(6*pi*x)}
xx=seq(0,.75,length=1000)
plot(xx,g(xx),type='l',lwd=3,col="red",xlab="x",ylab="g(x)")
```



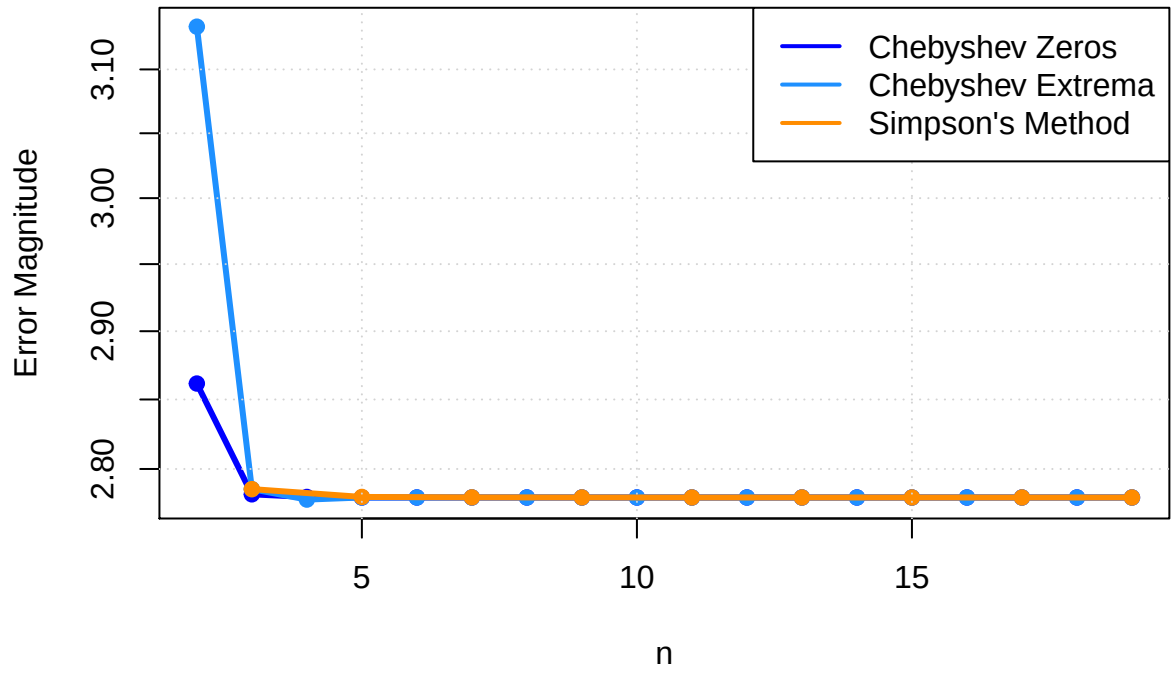

```
integration.test(g,0,.75,19,-1/(3*pi))
```



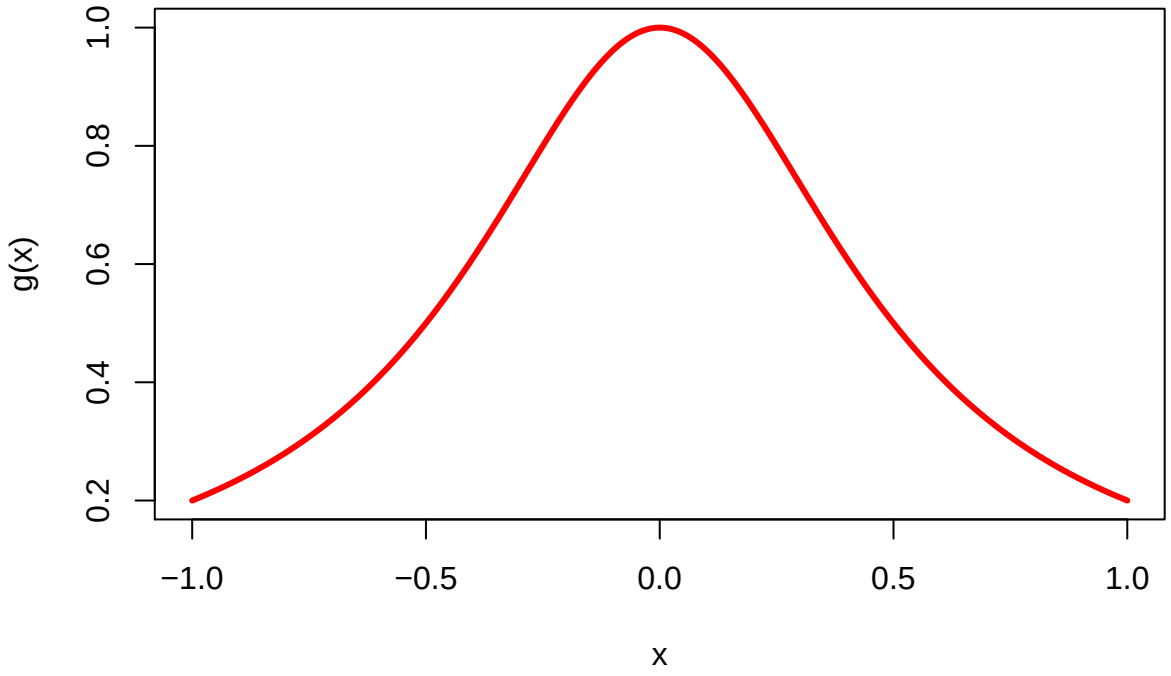

```
g=function(x){3*x^4+2*x^2+3}
xx=seq(-1,1,length=1000)
plot(xx,g(xx),type='l',lwd=3,col="red",xlab="x",ylab="g(x)")
```



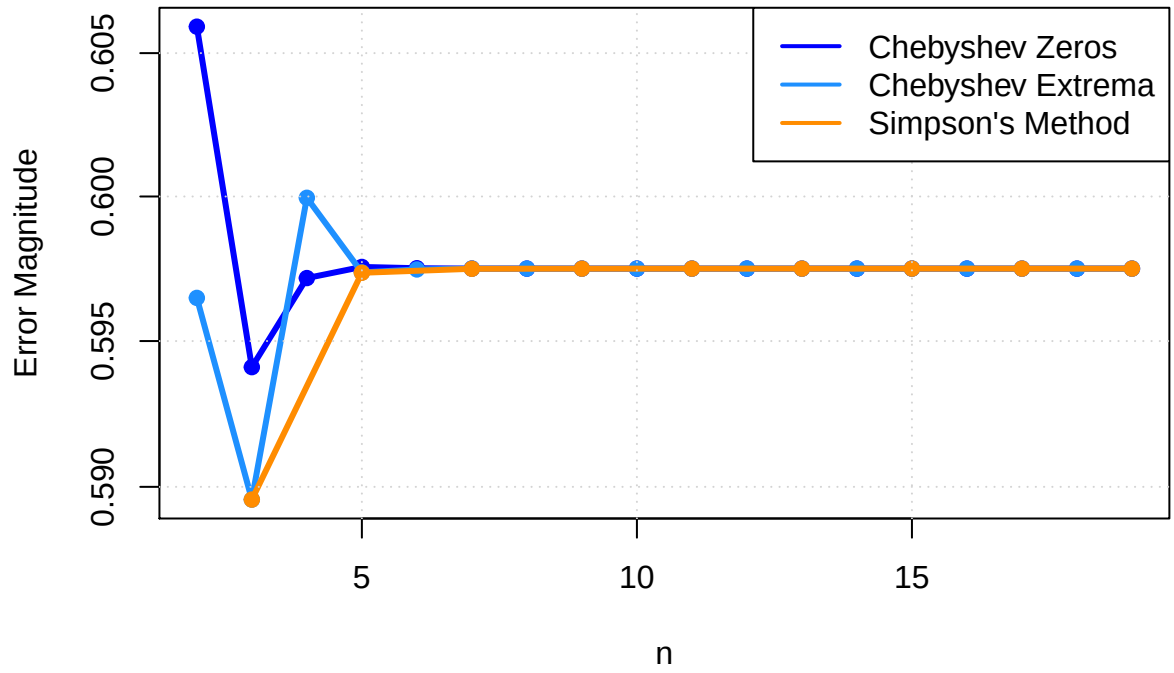

```
integration.test(g,0,.75,19,-1/(3*pi))
```



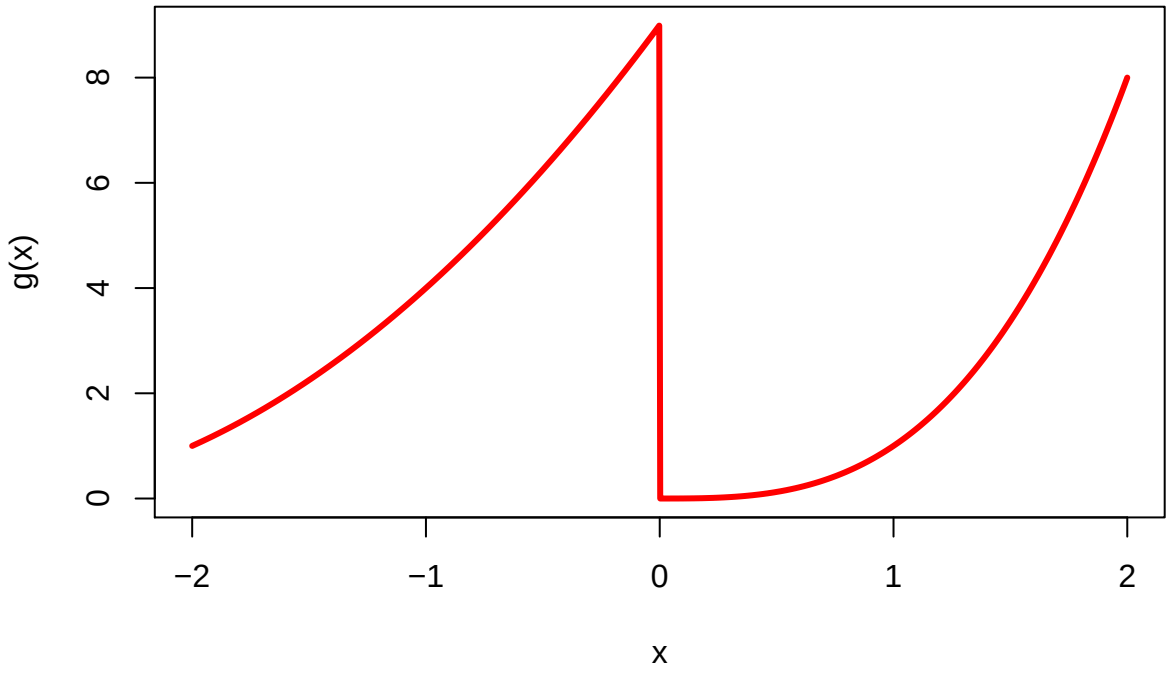

```
g=function(x){1/(1+4*x^2)}
xx=seq(-1,1,length=1000)
plot(xx,g(xx),type='l',lwd=3,col="red",xlab="x",ylab="g(x)")
```



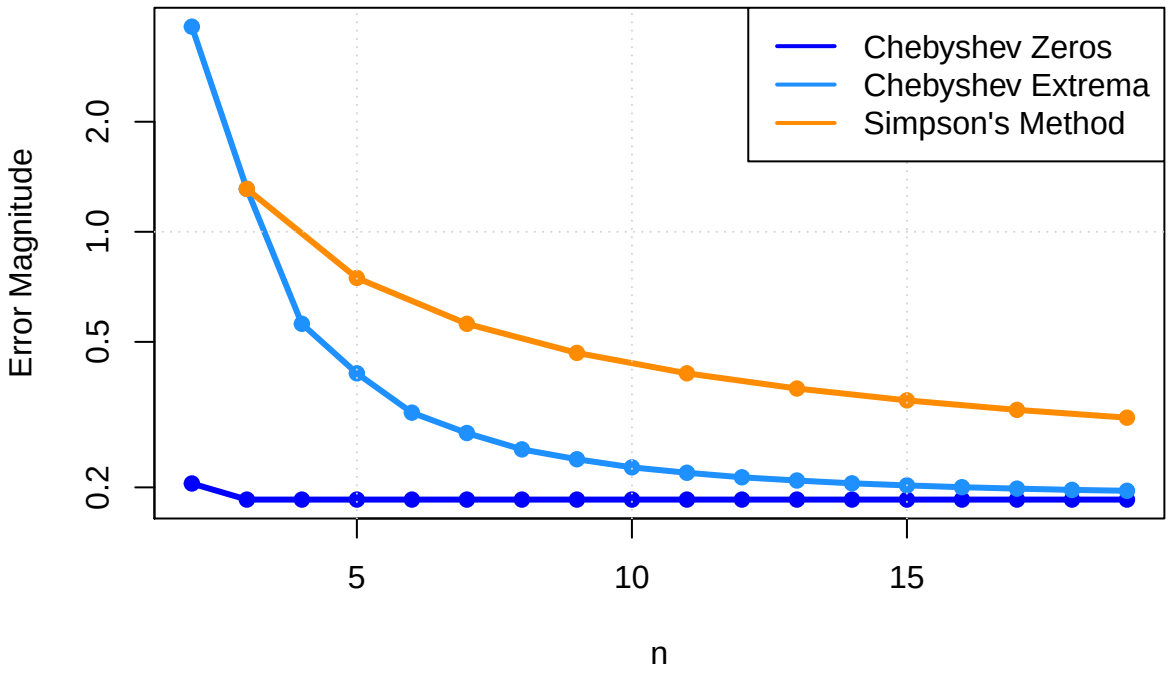

```
integration.test(g,0,.75,19,-1/(3*pi))
```




```
g=function(x){(x<=0)*(x+3)^2+(x>0)*x^3}
xx=seq(-2,2,length=1000)
plot(xx,g(xx),type='l',lwd=3,col="red",xlab="x",ylab="g(x)")
```




```
integration.test(g,0,.75,19,-1/(3*pi))
```



Choose some integrals that are interesting to you and test out the performance of the three methods. How do they compare? Is one always the best? Try it for a function that is not as smooth. How does the smoothness affect the convergence? Do you run into computational problems if you try to make n too large? If so, this is due to our use of the Vandermonde instead of the more efficient implementation discussed in the footnote above. You do not have to answer all of these questions comprehensively. I just want you to try a few examples to provoke some thought about the different methods, as well as to double check that your code is working correctly.

> Overall the Chebyshev method is generally better than Simpson's method, since the error decreases faster, and Chebyshev method converges faster than Simpson. Also, it seems like the more smooth the function is, the faster the error will converge.

Problem 6: Reflection on Technical Report 2

Please fill out the following Technical Report Reflection Form for TR2, and attach it to your homework. I would like the reflection to be detachable from your homework assignment.